1985

A programming environment for real-time control /

Mark Del Giorno
Lehigh University

A PROGRAMMING ENVIRONMENT

FOR REAL-TIME CONTROL

by

MARK DEL GIORNO

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Manufacturing Systems Engineering

Lehigh University

1985

CERTIFICATE OF APPROVAL

This thesis is accepted and approved in partial fulfillment of the requirements for the Degree of Master of Science.

12/14/85

(date)

Professor in Charge

Director, MSE Program

CSEE Department Chairman

ACKNOWLEDGEMENTS

I would like to thank Mary Lynn Fitzgerald and Tony Barbera for their continued friendship and guidance throughout the development of this thesis

I would also like to thank Roger Nagel for providing me with the means and opportunity to accomplish the work documented here and his many hours of proofreading and suggestions.

Most importantly, I wish to thank my wife, Sherry, for her loving support and understanding during the past year.

TABLE OF CONTENTS

ABSTRACT

Along with the advancement of Technology and the need for Computer Integrated Manufacturing, comes the demand for smarter, more flexible controllers at the process level with the ability to provide intelligent feedback information to higher levels within the factory. This thesis describes the driving principles behind a next-generation real-time control system originally developed by the Industrial Systems Division at the National Bureau of Standards, and the procedure by which that control system was advanced, re-developed and implemented at Lehigh University.

The control system provides a programming language (called SMACRO) and an environment in which application programs may be written, debugged and verified. The integration of the language and the environment allows for several features not found in conventional programming environments: An interactive capability is provided so part of an application program may be executed immediately from the terminal. A software modularity feature is also provided to greatly reduce the time spent in the debugging and verification stages. It is accomplished through the addition of an incremental compiler-interpreter which allows procedures (or sub-procedures) to be updated without the need to re-compile or re-link the entire application. The system

also has the ability to maintain and easily retrieve user-specified relationships within an application. For example, given a program that alters a certain set of variables, it is trivial to create a structure which relates those variables with that program so each time the program is executed, the values of the variables are printed at the terminal.

The SMACRO language and environment were enhanced from their original implementation at the National Bureau of Standards on Intel 8086 microcomputers to a Motorola 68000-based system. This thesis describes the development system environment and the basis for changes and upgrades to the original system.

Details as to the syntax of the SMACRO language and the system commands needed to manipulate the environment are also documented in a user's manual format.

CHAPTER ONE

INTRODUCTION

This thesis serves to document the work done at Lehigh University to develop and implement a non-conventional programming language/environment for real-time control. Many of the concepts discussed here evolved from the Industrial Systems Division of the National Bureau of Standards over the past decade.

This thesis is divided into five more chapters. Chapter Two identifies the characteristics of the environment to be created as well as the justification for choosing those particular characteristics.

Chapter Three identifies the specific steps taken to implement the environment and programming language outlined in the first chapter. A brief overview of the development system chosen for the implementation is provided.

Chapter Four explains the acutal syntax used to develop programs for a specific application. The different control structures are examined, and many examples are provided to suggest programming techniques.

Chapter Five shows how the environment is used during the program development and debugging stages of an application. It is the tools identified here that serve to integrate the attributes described in Chapter Two with the language of Chapter Four.

Finally, Chapter Six discusses the shortcomings of the system and provides several suggestions for possible future expansion of the system that may correct those deficiencies.

CHAPTER TWO

THE DEFINING ARCHITECTURE

By taking advantage of a closely coupled relationship with the originators of the real-time control system (RCS), at the National Bureau of Standards (NBS), this chapter will describe what the RCS accomplishes and how it differs from many conventional programming environments.

The RCS is comprised of two inter-dependent parts. The first is a programming language called SMACRO, so named because the programmer's primary means of writing application code is via a command set of system macros. The syntax of the SMACRO language is explained in detail in Chapter Four.

The second part of the system is the programming environment (operating system) in which SMACRO programs can be written, debugged, and executed. This environment is described fully in Chapter Five.

The language and the environment are integrated together to accomplish

four rather broad goals:

Interactive Capability

Software Modularity

Knowledge Based Representation

Debuggability

The reader will notice that some of the above desired features can in fact be found in some existing conventional programming environments, however it is the integration of these features with each other as discussed below that makes SMACRO and its programming environment unique.

Interactive Capability

Interactive capability allows the programmer of a particular application to use the system in a fast, convenient way. The programmer always has immediate access to any part of the SMACRO code associated with an application; so that at any time, a program, or even a portion of a program, may be executed by typing its name at the terminal without the need for the traditional compile-link-execute pipeline.

This interactive capability is accomplished by means of a next-generation interpreter which consults and maintains an on-line database. Each SMACRO construct (program or data type) defined by the programmer exists as an entry in this database along with other pertinent information associated with that particular construct. For example, a procedure's entry may contain several pieces of information: the name of the procedure; the object code associated with that procedure; the location of the source code on the disk; and perhaps

some other information to be discussed later in this chapter.

This on-line database is more commonly referred to as a dictionary, and each SMACRO construct is considered to be a word in that dictionary.

It is now easier to see how the interactive capability is accomplished. When a word is typed at the terminal, the dictionary is consulted to determine whether that word exists, and if so, how it should be processed by the system. For example, if the word is identified as a procedure, the object code associated with it can be executed immediately. Note that no limitation exists as to the level of the procedure to be executed. Main procedures as well as sub-procedures are all available for immediate execution.

In addition to the ability to execute programs, the environment allows the ability to query the dictionary as to the current state (value) of anything in the system: variables, constants, interface data, etc.

Software Modularity

Software modularity is the ability of the user to substitute new, updated software components for old components in a natural, easy to use environment.

There are three basic rules for modularity:

1. The new component must "understand" the format of the input data.

2. It must be capable of generating the proper output data in the correct format.

3. It must (at a minimum) perform the same function as the original component.

Given the previously defined dictionary structure, software modularity is easily achieved within the system. At any time a user may change the source

SMACRO code of a procedure or sub-procedure and "re-load" it into the system. The environment realizes that this word is already defined in the dictionary, so instead of creating a new entry, the old "definition" is updated with the new information provided. Many advantages of software modularity now become apparent.

First, when a change is made in a portion of an application, only that portion needs to be re-loaded into the system. There is no need to re-compile and re-link the entire application. All procedures which call sub-procedures do so via a reference to its definition in the dictionary, so when a sub-procedure's definition is changed, those procedures which call it will now reference its new definition automatically.

A second advantage of software modularity is its tendency to encourage the user to make minor (aesthetic) adjustments in procedures due to the quick response time associated with re-loading a single procedure and re-executing the application. These adjustments may be discouraged in conventional environments because of the prohibitive turn-around time in the re-compile and re-link process.

Lastly, given this software modularity feature, hardware modularity now becomes easier and more practical to implement. For example suppose a sensor fails and an identical one is not available for replacement. If a different one is available that performs the same function, the software drivers can be written in SMACRO off-line and loaded on top of the current system to incorporate the new sensor without the need to re-load the entire application. So "selective loading" of application procedures may take place depending on the hardware available at any particular time.

Knowledge-Based Representation

The objective of a knowledge-based representation is to maintain information about the behavior and inter-relationships of the programming environment and the application programs themselves. Some of this knowledge already exists in the system and some is supplied by the programmer. Both can take on many forms as indicated here.

The knowledge which already exists in the system is mainly information as to how different types of words execute depending on the context in which the programmer uses them. As we shall see in Chapter Five, the system may be placed into several different modes to alter the behavior of the SMACRO words in the dictionary. For example, in the normal operating mode of the system, when the entered word is a procedure, the system will execute the object code associated with that procedure, but in another (locate) mode, instead of executing that procedure, the system will display its source code and give the programmer the opportunity to change it.

The programmer's knowledge about the application is also maintained in the dictionary and the system provides tools which allow this information to be extracted later in a convenient way.

A primary vehicle for storing knowledge about the application is provided through the ability to define relational data structures. The most common relationship allowed by the environment is an owner-member (parent-child) relationship. For example, all variables in an application must have a user-specified owner known as (appropriately enough) a variable owner. In this way, the user may define related variables together so they will be members of

the same variable owner. The user can now interactively determine the state of those variables simply by referencing a single name - that of the variable owner, or by only remembering the name of a single variable, he can ask to see the other variables which are defined under the same variable owner.

Again, the specific resources that allow the programmer to query the relationships in the system are detailed in Chapter Five.

Debuggability

Debuggability is a feature of the environment which extends from a combination of the previous goals. Since the system is interactive, the programmer can quickly probe the system to determine the current values of variables or execute low-level procedures to insure that they are working properly.

Software modularity also implies debuggability to a certain extent, since the response time of the system to small changes is not prohibitive as in many conventional programming environments. Incremental changes can be made and evaluated until the application is finely tuned to the programmer's needs. If the integrity of a procedure is in doubt, diagnostic statements (such as "print") may be added, then that procedure can be re-loaded and tested again very quickly.

The knowledge-based representation provides for several operating modes of the system, one of which is a diagnostic mode. Each word in the dictionary when executed in this mode will respond with diagnostic information about itself depending on its type. Also, the previously mentioned locate mode provides easy access to the source code associated with any word in the system

be referencing only its name.

Summary

Now that the reader has been introduced to the basic architecture of the environment initially created at NBS, Chapter Three will identify the process by which that environment was created here at Lehigh, what modifications were made and why.

CHAPTER THREE

THE IMPLEMENTATION PROCESS

The National Bureau of Standards expressed a desire to develop a next-generation Real-Time Control System based on the concepts discussed in Chapter Two. Lehigh University was identified as the institution best qualified to perform the necessary work, and a grant was issued to Dr. Roger Nagel, Director of the Institute for Robotics.

This chapter discusses the process by which the enhanced version of RCS was implemented at Lehigh University. The implementation has been divided into five basic steps:

1. Analysis of the Current Implementation at NBS
2. Selection of an Appropriate Development System
3. Development of SMACRO and its Operating Environment
4. System Verification and Debugging
5. System Documentation

This thesis is the primary vehicle by which the fifth step was accomplished. The other four are discussed below.

Analysis of the Current Implementation at NBS

The development system hardware on which RCS runs at NBS consists of several Intel 8086 microcomputers in an Intel Multibus^{tm} environment. Each of the microcomputers has its own operating system and can operate independently of the others. The Multibus^{tm} provides a common memory accessible by each microcomputer, and thus supplies a simple form of communication between them.

The low-level software used to create SMACRO on the development system is Forth^{tm}. It was chosen for several reasons which parallel the concepts conveyed in Chapter Two.

First, it is a highly interactive operating system as well as a programming language. Second, its fundamental mechanism for defining new constructs (words) is through the use of a system dictionary very similar to the one described in Chapter Two. Forth^{tm} is also extensible. That is, the tools exist to allow a user to define new constructs which can add to or modify the system dictionary rather easily.

A major negative aspect of the NBS version of RCS is the lack of sufficient addressable memory to develop large application programs. This is due to a conflict involving Forth^{tm} and the segment registers of the Intel 8086 microprocessor, and has been identified by the original RCS developers at NBS to be a major obstacle in the expansion of the current system.

Selection of an Appropriate Development System

In choosing a development system, it was apparent that the Forth[tm] language was ideal given the structure imposed upon the system (as described in Chapter Two) but the insufficient memory problem was one that could not be tolerated.

Fortunately NBS had identified a new commercially-available development system (the SBE-200) which supports both Forth[tm] and the Multibus[tm] environment on a more powerful 32-bit processor, the Motorola 68000. This processor would increase the maximum application size by a factor of eight.

Thus the decision was made to purchase the SBE-200 system and use it to develop the next-generation of RCS at Lehigh. It should be noted that Forth[tm] programs are not generally transportable between computers, so it was not possible to simply transfer the existing Forth[tm] programs onto the new machine. It was also not desirable to do so, since much of the additional computing power provided by the Motorola 68000 would not be incorporated.

Since Forth[tm] has become the operating system in which SMACRO programs are written as well as the core of the RCS programming environment, a brief overview of Forth[tm] is provided here. For the reader interested in learning how to program in Forth[tm], Starting Forth by Leo Brodie, Prentice Hall, 1981 is highly recommended as an excellent reference.

As mentioned previously, Forth[tm] maintains a global dictionary where the definition of every word in the system is kept. Initially the dictionary contains only Forth and SMACRO reserved words. In order to add application

programs to the dictionary, the programmer creates those programs on the disk, and then "loads" them into the system.

The means by which the disk is maintained in Forth[tm] is rather unique. It is divided into individual units called "blocks" in which programs are stored. Each block contains 1024 characters of source text divided into 16 lines of 64 characters each. The blocks are numbered consecutively from 1 to 32,768. A typical block may look like this:

```
10000 LIST
    0 % THIS IS A TYPICAL BLOCK
    1
    2
    3  I could put a SMACRO program here if
    4   I wanted to!
    5
    6
    7
    8
    9
   10
   11
   12
   13
   14 Blocks are never longer than 16 lines!
   15
```

To see the contents of any other block in the system, the **LIST** command would be used as follows:

<block-number> **LIST**

A screen editor exists which allows the contents of a block to be easily manipulated. To invoke the editor use the **ED** command:

<block-number> **ED**

The editor was developed at NBS by John Michaloski, and a copy of unpublished documentation is available through the Office of the

Manufacturing Systems Engineering Program at Lehigh University.

A special note about blocks: Forth[tm] denotes the last block to be **LIST**ed or **ED**ited as the current block and "remembers" its number, so if the block number is not specified before the **LIST** or **ED** commands, the current block will be assumed.

After a program's definition has been saved in a block on the disk, the programmer can have that information be entered into the system dictionary by using the **LOAD** command:

<block-number> **LOAD**

Again, if the block-number is omitted, the current block is assumed.

Blocks are restricted to 16 lines to encourage the programmer to develop several small modular sections of code which accomplish very specific tasks, and thus make debugging easier.

Three reserved words commonly used during the loading process are **THRU**, %, and %%. **THRU** is uded to load a sequence of blocks in order. Its syntax is

<start-block-#> <end-block-#> **THRU**

and it loads start-block-# through end-block-# inclusive.

The word % is used within a block as a signal to the loading mechanism that the rest of the line on which % appears is to be treated as a comment.

Likewise, %% indicates that the rest of the current block is to be ignored during the loading process.

An application usually consists of a collection of blocks which are **LOAD**-ed together. In this case it is useful to make a single block which loads several other blocks. For example

```
9000  LIST
 0
 1 9001 LOAD    % procedure "a"
 2
 3 9002 LOAD    % interface procedure
 4
 5 9003 LOAD   % Initialization procedure
 6
 7 9004 LOAD
 8
 9 9006 9020 THRU    %% REST OF APPLICATION
10
11
12  This may be used for general comments.
13
14
15
```

Now by entering 9000 **LOAD**, all the above blocks will also be loaded.

The situation occasionally arises when the application programmer wishes to clear the dictionary of all the current application words and begin anew. This is accomplished by the word **EMPTY**. **EMPTY** resets the dictionary to its original power-up condition, and a new application can then be loaded.

Development of SMACRO and its Operating Environment

Prior to the creation of SMACRO it was necessary to develop a complete understanding of the SBE-200 system, and an extensive knowledge of the 68000 microprocessor instruction set. Once this understanding was achieved, it was possible to begin implementation.

It was desired to have the SMACRO language appear in a form similar to that of many conventional high-level languages such as Pascal and C in order to facilitate programmability by the unfamiliar user. The implementation of

SMACRO involved the development of a large number of system constructs with the ability to parse SMACRO programs and generate the corresponding assembly code without the need for a conventional linker (the code will reside in the Forth^{tm} dictionary). The result of this implementation is the constructs and operators identified in Chapter Four.

The biggest challenge in the re-development of RCS was the modification of the Forth^{tm} environment to achieve the desired attributes identified in Chapter Two.

There were three major modifications/additions necessary:

1. Development of System Modes
2. Implementation of Software Modularity
3. Creation of Member-Owner Relationships

The development of the different system modes allows the definition of each word in the dictionary to be interpreted differently based on the current (user-selectable) mode in which the system is operating. The types of modes available are fully discussed in Chapter Five.

The implementation of the software modularity aspect was a rather complex modification. Forth^{tm} did not provide the programmer with the ability to alter the definition of an existing word in the system dictionary without re-booting the system and re-loading the entire application. This problem was solved through the addition of a level of indirection, so each word is now associated with a pointer to that word's current definition. When the definition of a word is to be changed, the new definition is loaded into available memory, and its corresponding pointer is simply re-directed to point to the new definition.

The Forth[tm] environment also had to be expanded to allow the programmer to create member-owner relationships, and the tools were provided so those relationships could be retrieved and used in a convenient way. Several examples as to the usefulness of these structures are provided in Chapters Four and Five.

Other modifications and additions to SMACRO and its environment were identified as a result of several joint meeting involving Lehigh and NBS personell. These included increased error detection, type checking and additional data types.

System Verification and Debugging

This stage was an ongoing iterative process concurrent with the actual development of SMACRO and the development of its operating environment. Many test procedures were written to insure the operational integrity of the system.

In addition to confirming the performance of the system in "normal" conditions, extensive diagnostics were also developed to ascertain whether error detection and recovery was satisfactorily achieved at both compile-time and run-time.

Chapters Four and Five will now identify the specifics of how the environment is programmed and used respectively.

CHAPTER FOUR

PROGRAMMING IN SMACRO

This chapter explains in detail the syntax associated with the SMACRO programming language. It does not require the user to have a working knowledge of Forth, however an understanding of some high-level language (e.g. Pascal or C) will be assumed.

Identifiers

An identifier is any user-defined word that performs a specific function. For example, names of variables, programs, subroutines, etc. Each identifier is represented as an entry in the system dictionary and can be considered "global." All SMACRO identifiers must satisfy three conditions:

1) They must contain less than 32 characters.

2) They must be unique in that the same identifier cannot be used twice nor can reserved words be used.

3) They may contain any combination of the 127 ASCII characters excluding the blank which is used as a delimiter (separator between identifiers).

SMACRO is case-sensitive, so "fred" and "FRED" represent different identifiers.

Examples of valid identifiers:

```
test-routine
test_routine
#chars/STRING
?READY
!@#$%^
```

The reader should notice that identifiers need not begin with a letter, and more importantly, they need not contain letters at all. This brings up an important peculiarity of the SMACRO language. It is possible for the programmer to create an identifier that consists only of the digits 0 through 9, thus appearing to be a number. For example, an integer variable could be assigned the name: 12345678 and this "name" would be associated with a variable value! It is for this reason that the definition of any identifier consisting only of numerical digits must be avoided at all times.

Numeric Constructs

Numbers are not allowed within SMACRO programs. They may only be

used during the declaration of data types (variables, constants, etc.) for initialization purposes. The attempt of this philosophy is to encourage the programmer to assign that number to an identifier with a meaningful name so the program is more readable.

The values of numerical constructs which represent numbers are stored as 32-bit signed integers, so they may take on values between -2147483648 and +2147483647 inclusive. However, many of the operations of the Motorola 68000 (the microcomputer on which SMACRO is written), such as multiplication and division, restrict the operands to 16 bits, so it is advisable to limit numbers to the range

$$-32768 < X < 32767.$$

Variable Owner Declaration

Defining variables and constants in SMACRO is similar to Pascal with one major exception. At the time of any variable's declaration, the user must designate a Variable Owner which will "own" that variable (and perhaps several others). The advantages of grouping variables together under a common owner are two-fold. First, the values of the "member" variables are stored in contiguous memory locations, so all the elements of a variable owner can be transported easily as a group. Second, the values of all the "members" of a variable owner can be examined simply by referencing the variable owner's name. A typical variable owner declaration appears as follows:

<#> VAR-O <var-o-name>

The <#> corresponds to the buffer size (in bytes) to be allocated for this

particular variable owner, **VAR-O** is the SMACRO defining word, and <var-o-name> is the name being assigned to this particular variable owner.

Data Construct Declarations

All data constructs must be declared before being used in a program. Remember, all constructs defined are global and accessible form any procedure or sub-procedure in the application program. The syntax for defining each of the different constructs currently offered by the system is discussed here:

iv <name>

Declares <name> to be a 32-bit integer variable. An initial value is not assigned to the variable by the SMACRO environment.

bv <name>

Declares an 8-bit byte variable. These are often used to store single characters or to read data from an input port.

c con <name>

Declares a 32-bit constant with value "c". Once a constant has been declared in the system its value cannot be changed.

i 1:a <name>

Declares a one-dimensional array of "i" 32-bit words. The elements of the array are accessed with index 0 through index i-1. Once declared in the system, the size of the array may not be changed. The reference to an element of an array is made as follows:

<array-name> { <index> }

The index may be any construct which represents an integer (**con**, **iv**, another **1:a** etc.) Again, remember that spacing is very important in SMACRO. The "{" and "}" must be delimited by spaces to be recognized by the system.

ptr <name>

Declares a pointer to a 32-bit integer value. When the pointer is used in an expression, a double indirect addressing scheme is employed so the data used is the contents of the address to which the pointer is pointing.

To reset the **ptr** to point to a particular memory location, the **reset-ptr** command is used within a routine:

reset-ptr <value>

The <value> is commonly a constant which specifies a memory location.

strv <name>

Declares a 16 character string variable. Strings may be used in two ways within the system: The value of one string may be transferred to another string, or strings may be tested to determine equality. Strings may not be added to or compared with any other data type.

All data constructs declared will be "owned" by the last variable owner to be defined, thus each will be allocated an appropriate number of bytes in that variable owner's buffer depending on the construct type.

An example of a typical variable declaration statement:

```
100 VAR-O  Example-var-owner
      iv     sample-integer-var#1
     strv    sample-string-var#1
300   con    terminal-i/o-address
 10  1:a     simple-array
      ptr     pointer-to-something
```

A note about numbers appearing on the left hand side of a declaration: those numbers are not subject to the software modularity feature described in Chapter Two. Once the size of a variable owner, the value of a constant, or the size of an array is declared, it cannot be changed except by re-starting the system and re-loading the entire application.

The programmer is therefore encouraged to be generous with the byte allocation for a **VAR-O** or a **1:a**. Also, if it is anticipated that the value of a **con** may need to be changed, it should be defined as an **iv** with the appropriate initial value as described in the next section.

A convention has been established for identifiers that represent constants which cannot be assigned a meaningful name. The identifier should consist of the number followed by the # character. The numbers 0 through 9 are defined in this way for the convenience of the programmer:

```
40 VAR-O NUMBERS
    0 con 0#     1 con 1#     2 con 2#
    3 con 3#     4 con 4#     5 con 5#
    6 con 6#     7 con 7#     8 con 8#
    9 con 9#
```

Remember, these should only be used if a suitable name cannot be substituted. The reader should notice that the above **VAR-O** is full. That is, 40 bytes were allocated for its buffer, and ten constants were defined of four bytes each.

Data Construct Initialization

The user has the ability to initialize all variable data constructs at the

time of their declaration. This is done using the backward store operator: <= after the declaration. The general syntax is as follows:

<declaration> <= <initialization-value(s)>

The initialization values may take the form of literals (numbers) or SMACRO constants only. No expressions are allowed. Examples are provided below:

```
100 VAR-O test-var-o
  1024 con    one-kilobyte
       iv     Integer-var     <= 8
    5  1:a    Array#1         <= 0 1 #lines/page 3 4
       strv   string-var      <= "testing123"
       iv     Integer#2
```

There are several important points to note about the initialization operation:

- o Constants may not be initialized using <=. Their value is determined only by the number preceding **con**.

- o The initialization is optional, however, the user should never assume a variable is initialized to zero (or any other value for that matter) at the time of its declaration.

- o If an array is to be initialized, a value must be provided for all elements in the array.

- o Even though strings are 16 characters long, the initialization string may contain fewer characters (surrounded by quotation marks). The remainder of the string will be filled with blanks.

- o If an identifier is used on the right hand side of the <= operator instead of a number, it must have been previously defined.

Numerical Expressions

SMACRO's numerical expressions differ from those of conventional programming languages. The expression to be evaluated is found on the left of the assignment operator, and the destination where the result of the expression is to be placed is found on the right.

The arithmetic operators used are (+), (-), (*) and (/). There is no precedence - all expressions are evaluated from left to right. Notice that each operator consists of three characters type together without spaces.

The expression

```
A (+) B (*) C => D
```

adds A to B, then multiplies that result by C, and stores the result in D.

Several important points about expressions:

- o Literals (numbers) are not allowed. All elements of numerical expressions must be represented by an identifier in the system.

- o Spacing is very important - since SMACRO allows identifiers to consist of any ASCII characters, the user must provide at least one space before and after each identifier.

- o The assignment operator is two characters used together: "=" and ">" written as =>.

Routines

The *routine* is the primary building block for creating SMACRO programs. It is composed of both numerical expressions (discussed above) and control statements (to be discussed later in this chapter).

The syntax for defining a routine is

```
routine  <routine-name>
    ...
    (expressions and control statements)
    ...
end-routine
```

Member-Owner Constructs

Two constructs other than the variable owner exist which allow the user to group several SMACRO words under a single owner for easy access (in debugging, for example). They are the executing owner (**EXEC-O**) and the list owner (**LIST-O**).

The syntax for declaring an executing owner is

```
EXEC-O <executing-owner-name>
    <routine-1>
    <routine-2>
    <routine-3>
    ..........
END-EXEC-O
```

An executing owner may be thought of as a routine that only has the ability to call other routines. In the normal operating mode of the system, an EXEC-O will call the routines it owns in order.

The syntax for declaring a list owner is

```
LIST-O <list-owner-name>
    <SMACRO-identifier#1>
    <SMACRO-identifier#2>
    <SMACRO-identifier#3>
    ..............
END-LIST-O
```

The identifiers can be of any type: **iv**, **routine**, **EXEC-O** etc. Even other

LIST-O's are allowed. A list owner serves no function as far as executing anything. It is simply a means of grouping SMACRO constructs which can be conveniently accessed through a single word - that of the list owner.

See chapter Four as to how executing owners and list owners behave in the different modes of the system.

Conditional Statements

Several control structures are available in SMACRO for testing and branching. All of them use condition statements. A condition statement evaluates to being "true" or "false" depending on the values of the arguments. The comparison operators are listed below. Again the reader should note that the parentheses are a part of the operator.

Comparison operators:

(EQ) - Equal to
(NE) - Not equal to
(LT) - Less than
(LE) - Less than or equal to
(GT) - Greater than
(GE) - Greater than or equal to

Conditional expressions are in the form of

<data construct> <comparison operator> <data construct>

Several expressions can be combined with conjunctions for multiple testing.

The possible conjunction operators are as follows:

(AND) - Both conditions must be true
(OR) - Either (or both) condition must be true
(XOR) - Exactly one condition must be true

Conditional expressions which contain conjunctions are evaluated as follows: each individual conditional expression is evaluated to be true or false as if the conjunctions were not present. Then the conjunction operators are applied to these true/false values from left to right to determine an overall truth value.

As an example, consider the following relatively complex conditional statement:

a (EQ) b (AND) c (LT) d (XOR) e (GE) f

The three expressions

a (EQ) b
c (LT) d
e (GE) f

will each be evaluated, and then the truth value of the first expression will be "and-ed" with the truth value of the second, and that joint result will be "exclusive or-ed" with the truth value of the last expression.

Program Control Structures

Program control structures are used for conditional branching and looping within a routine. The traditional structures available include:

if...then...else...endif
while...do...end-do
repeat...until...end-repeat
case...case-var:...end-case

which are discussed below.

There is also a structure called a **state-table** which will be explained in detail later.

if...then...else...endif

Syntax:

```
if <condition-expression> then
    ...
    (SMACRO code for condition-expression=true)
    ...
else
    ...
    (SMACRO code for condition-expression=false)
    ...
endif
```

The else portion is optional. Also, SMACRO does not care what the format of the code looks like within a block, so the following portion of a routine is perfectly valid:

```
if a (EQ)
    b (XOR) c
    (NE) d
then  e (+)
    f => h else e (-) f => h endif
```

But it is preferable to input code in a more readable format:

```
if a (EQ) b (XOR) c (NE) d then
    e (+) f => h
else
    e (-) f => h
endif
```

while...do...end-do

Syntax:

while <conditional-expression> **do**
...
(code to be repeated)
...
end-do

As in conventional programming, the code within the loop may not be executed at all if the conditional-expression is initially false. A simple example of a while..do...end-do loop follows:

while a **(LT)** b **do**
a **(+)** 1# **=>** a
end-do

In this example, "a" will be incremented until it is equal to b, at which time control will exit the loop. If a is initially greater than or equal to b, it will not be changed (since the loop will never be entered).

repeat...until...end-repeat

Syntax:

repeat
...
(code to be repeated)
...
until <conditional-expression> **end-repeat**

The code within the loop will be executed at least once before the condition is checked. So the example

repeat

a (+) 1# => a

until a (GE) b **end-repeat**

will behave like the **while...do...end-do** example discussed earlier except if a is initially greater than or equal to b, it will be incremented once before dropping out of the loop.

case...case-var:...end-case

Syntax:

case <variable>
case-var: <test-var>
... (code to be executed) ...
case-var: <test-var>
... (code to be executed) ...
....
....
default:
... (code to be executed) ...
end-case

The **default:** is optional in the **case** statement, however it is highly recommended that it always be used, if for nothing more than to print a message that no match was found.

An error condition exists in a case statement under two situations. First, if no match is found, and a **default:** case is not specified, and second if two or more **case-var:**'s match the variable being tested. In either of these situations, program execution will be aborted, and the user will be informed as to which occurred.

An example of a case statement follows:

```
case  a
case-var: b
        PRINT" a matched b"
case-var: c
        PRINT" a matched c"
case-var: d
        PRINT" a matched d"
default:
        PRINT" a did not match anything"
end-case
```

For the impatient reader, the **PRINT"** statement is discussed at the end of this chapter.

State-tables

State-tables allow the user to make decisions based on the values of up to three variables simultaneously. As an example, suppose we have two integer variables that can each take on one of two values: 1 or 2. There are four possible "states" that can occur:

state #1: a = 1, b = 1
state #2: a = 1, b = 2
state #3: a = 2, b = 1
state #4: a = 2, b = 2

Any other possibilities must be considered an error, since it would imply that either a or b has a value other than 1 or 2.

Now suppose we want to call one of four procedures depending on the combined state of a and b. Conventional programming would dictate using either nested **case...end-case** statements, or nested **if...then...else** statements. I'll choose the latter for illustrative purposes:

```
if a (EQ) 1# (AND) b (EQ) 1# then
    procedure-1
else
    if a (EQ) 1# (AND) b (EQ) 2# then
        procedure-2
    else
        if a (EQ) 2# (AND) b (EQ) 1# then
            procedure-3
        else
            if a (EQ) 2# (AND) b (EQ) 2# then
                procedure-4
            else
                PRINT" Invalid State"
            endif
        endif
    endif
endif
```

Even though the above code will perform the function desired, it has (at least) two shortcomings. First, it is difficult to debug. Knowing which procedure was executed, it is not trivial to look at the code and identify what values of a and b generated the call to that procedure. Second, as the number of variables increases linearly, the complexity of the code increases exponentially, and it quickly becomes unmanageable. State-tables provide a means to perform the same decision function as the nested **if...then...else** statements using a more convenient notation. The state-table format which performs the equivalent function as the **if...then...else** example above is shown here:

```
state-table EQUIVALENT
    vars:     a          b
    cond:  (EQ) 1#    (EQ) 1#    procedure-1
    cond:  (EQ) 1#    (EQ) 2#    procedure-2
    cond:  (EQ) 2#    (EQ) 1#    procedure-3
    cond:  (EQ) 2#    (EQ) 2#    procedure-4
    no-match: PRINT" Invalid State"
end-statetable
```

Notice how clear and compact the state-table is. The function being represented here is much easier to understand. The general form for a state-table is shown below:

```
state-table <state-table-name>
   (pre-process code)
   vars:     <var-1>      <var-2>      <var-3>
   cond:  <condition>  <condition>  <condition>  <code>
   cond:  <condition>  <condition>  <condition>  <code>
   cond:  <condition>  <condition>  <condition>  <code>
   cond:  <condition>  <condition>  <condition>  <code>
   .....  ...........  ...........  ...........  ......
   no-match: <default-code>
end-statetable
```

There are several points to be made about state-tables:

- o The pre-process section may consist of any expressions/constructs that can be used inside a routine definition (calls to other routines, numerical expressions, control branching, etc.). It is generally used to reduce the information present in several variables down to three so the **state-table** can be used.

- o State-tables are defined in SMACRO as independent entities (like routines). They are callable from routines as well as other state-tables. They are not recursive (they cannot call themselves).

- o <var-2> and <var-3> are optional. If neither is present, the state-table will behave very much like the single variable **case** statement except the user will have the opportunity to check for conditions other than equality.

- o <code> is the SMACRO code to be executed if that particular "line" of the state-table matches during execution. It may be a routine, a list of several routines, or even another state-table. It may also consist of arithmetic expressions, however this is discouraged due to lack of modularity. A common programming practice is to use an **EXEC-O** as the <code>.

- **no-match:** is optional, but its use is to be encouraged. If it is not present and no lines in the state-table match the input state, then execution will be aborted and the user will be informed of the error.

- If more than one line matches, execution will stop and the user will be informed of the error.

Other SMACRO Constructs

Several other constructs exist in SMACRO to perform various functions:

PRINT"
PRINTCR"
from-stack
to-stack

These are discussed below.

PRINT"

PRINT" is a method of printing text to the terminal without generating a carriage return. It may be used within any routine or state-table.

Syntax:

PRINT" <text to be printed> "

Note that the beginning quotation is part of the construct, and the end quotation need not be separated by a space from the text to be printed. Any attempt to print quotation marks within the quotation marks will not work.

PRINTCR"

PRINTCR" is used exactly as **PRINT"** except that after the line is printed, a carriage return and line feed will be generated.

Syntax:

PRINTCR" <text to be printed> "

from-stack

from-stack provides a means to pop data off the parameter stack and use it within a SMACRO routine. (This is a common way of passing data between routines.) It is used just as any integer variable would be inside a routine. A typical use would be as follows:

```
a (+) from-stack (/) c => b
```

or even

```
if from-stack (EQ) 0# then
    PRINT" Top stack element is zero"
else
    PRINT" Top stack element is non-zero"
endif
```

to-stack

to-stack allows SMACRO routines to put values on the parameter stack. It is used in place of the store operator in an arithmetic expression:

```
a (/) b (-) c to-stack
```

To double the value of the number currently on the stack:

```
from-stack (*) 2# to-stack
```

Summary

This chapter has defined the syntax in which SMACRO programs are to be written. Chapter Five will show how the system is to be used and debugged once an application is written and loaded into the environment.

CHAPTER FIVE

HOW TO USE THE ENVIRONMENT

This chapter describes the programming environment in which SMACRO programs are written, debugged, and executed. The specifics are explained to show how the user can take advantage of the system features identified in Chapter Two during application programming.

The reader should recall from Chapter Three that the SMACRO programming environment is essentially an extension of the Forth[tm] environment.

The reader should also recall that Forth[tm] provides the user the ability to execute any defined entry (word) in the system dictionary directly from the terminal simply by typing its name. SMACRO enhances that capability through the addition of multiple code fields. Multiple code fields allow words to execute differently based on two things:

1. The type of word to be executed.

2. The current "mode" of the system.

The type of word to be executed may be any one of the constructs identified in Chapter Four: **routine, VAR-O, iv, state-table,** etc.

The mode of the system is user-selectable and offers one of the system's most powerful programming/debugging tools. Currently there are 3 possible operating modes of the system: run mode, locate mode, and diagnostic mode. The current mode of the system can be identified by the prompt at the terminal:

:R - Run Mode

:L - Locate Mode

:D - Diagnostic Mode

Each of these is discussed below.

Run mode

This is the "normal" operating mode of the system. It is entered by typing **:R** at the terminal. The prompt will now echo the **:R** back as long as the system remains in run mode. The run mode behaviors of several SMACRO types are listed here:

routine - Executes the code associated with it.

state-table - Executes as defined in Chapter Four.

EXEC-O - Executes each of the routines it owns in order.

LIST-O - Prints each member's name, type, and current value, if applicable.

The reader should keep in mind that the system modes are used as an interactive tool mainly to query the system, and only affect the behavior of words typed at the terminal.

Once a **routine**, **EXEC-O**, or **state-table** is invoked in run mode, it will behave exactly as described in Chapter Four, even if the mode of the system is somehow altered in the middle of that execution.

A special note for the Forth[tm] programmer: The SMACRO data constructs (**iv**, **con**, **1:a**...) also have a run mode behavior associated with them which is identical to their Forth counterparts (e.g. an **iv** leaves the address of its value on the parameter stack.)

Locate Mode

Locate mode is entered by typing **:L**. The system will appropriately respond by changing the prompt to **:L**. Locate mode dictates the same behavior for all data types: when a SMACRO word is executed in locate mode, the dictionary is consulted to determine from which block on the disk that word was defined. That block is then displayed at the terminal.

This feature reduces much of the burden on the user to remember specific block numbers. Once the application programs are loaded into the system, the user can access the definition of any word by referring to it by a (hopefully) meaningful name rather than a block number.

Diagnostic Mode

This mode gives the user extensive program debugging capability. It is entered by typing **;D**. As a minimum, each SMACRO word when executed in diagnostic mode prints its name with its type in parentheses. In addition, several types perform some additional functions:

VAR-O - Executes all the data constructs it owns in their respective diagnostic modes.

iv - Prints its current value.

con - Prints it current value.

1:a - Prints the values of all elements of the array.

strv - Prints the current value of the string.

ptr - Prints the address it is pointing to and the contents of that address.

EXEC-O - Prints all its members in order and identifies them as routines.

LIST-O - Executes each of its members in its respective diagnostic mode.

It is possible for the user to take advantage of the different modes of the system during the execution of a SMACRO **routine**. This is accomplished by

using the word **:execute** as follows:

:execute <mode> <SMACRO-word>

The mode can be either **:R**, **:L**, or **:D**, and the SMACRO-word may be any defined identifier in the system dictionary. For example assume a block is loaded that contains the following:

```
8 VAR-O some-vars
    iv   a
    iv   b

routine test-it
    update-a-and-b
    :execute :D some-vars
end-routine
```

When the routine test-it is executed in run mode, it will call the procedure labeled 'update-a-and-b' and then execute the **VAR-O** "some-vars" in diagnostic mode which will generate an output that looks something like this:

```
test-it (VO)
    a (iv) = 8
    b (iv) = 11   :R
```

Remember that **:execute** is only to be used within a **routine** (or **state-table**).

Another Interactive Capability

The last feature of an interactive environment mentioned in Chapter Two was the ability to execute any portion of an procedure directly from the terminal. This can be accomplished by using **PORTION**.

Syntax:

<start-line-#> <end-line-#> PORTION

PORTION will immediately execute the SMACRO code in the current disk block beginning with <start-line-#> up until <end-line-#> inclusive. A common usage of **PORTION** is to enter **:L** (locate) mode, type the name of a routine (so its source code will print, and it will become the current block), then use **PORTION** as shown above to execute a specific part of that routine. Note that **PORTION may only be used from the terminal**. Never inside a SMACRO **routine.**

Software Modularity

As mentioned in Chapter Two, the programmer has the ability to change any part of an application and re-load (re-define) only that part at any time. Two common conditions for re-loading are noted here:

- o When a routine is not executing properly, diagnostics can be quickly added and that routine can be re-loaded and re-executed immediately.

- o If the user wants to change the initial value of a data construct, the construct can be easily located (in **:L** mode) and a backstore operator (**<=**) can be added with an initial value, or if already present, the initial value can be changed and the block re-loaded.

During the re-load process the system will automatically return to :R mode and several error conditions will be monitored:

o An attempt to change an identifier's type. Example: An identifier was defined as a routine and it is being re-loaded as an integer variable.

o An attempt to change a number which appears on the left hand side of a declaration. Examples: The value of a **con** (if it were meant to be changed, it should have been defined as an **iv** and the backstore operator used for initialization); The size of a variable owner; The number of elements in an array.

If any of the above errors occur and the programmer wishes to make the change anyway (for example, re-define a constant as an integer variable), the application will have to be re-loaded entirely.

In addition to the error situations above, a warning message will be generated if a constant is redefined from a block different than its original definition. This feature is provided to aid the forgetful programmer who mistakenly uses the same name for what are supposed to be two different routines (or any other construct) in different parts of an application. This is not assigned an error status to allow for the programmer who is taking advantage of the hardware modularity feature described in Chapter Two.

Summary

This chapter identified the features of the system which satisfy the initial objectives of the system. Chapter Six contains some closing remarks about the future application of the Lehigh version of RCS.

CHAPTER SIX

CONCLUSIONS

This chapter is divided into three parts. First, the system additions and modifications incorporated into the Lehigh version of RCS are identified. Second, suggestions as to how the new SMACRO programming language and environment can be improved upon in the future are provided, and lastly some final comments are made as to the scope of the applications for which this next-generation controller can be applied.

System Additions and Modifications

Several alterations were made to both the SMACRO language and its environment when compared with the NBS version. These modifications were in the form of improved run-time and compile-time error detection as well as the addition of new data types and control structures. Each is specifically discussed here.

Two run-time error diagnostic algorithms were added to check the **case** and **state-table** control structures. In the NBS version, the first set of conditions to match the input variable(s) is executed without checking the remainder of the structure. The Lehigh version of RCS insures that multiple matches do not occur before executing an output procedure. If more than one match does occur, an error will be generated and execution will be stopped.

A compile-time diagnostic was added to insure that each identifier in the system is unique. Previously if the user mistakenly gave the same name to two different types (e.g. an integer variable and a routine), the system would not recognize the discrepency.

An existing compile-time diagnostic was altered to allow for the hardware substitution concept discussed in Chapter Two. In the NBS version of RCS, when the user attempted to re-load a routine from a block other than the one in which it was originally defined, an error message would be generated and compiling would cease. The Lehigh version generates a warning message to the programmer, but still updates the definition of the routine in the dictionary and continues compilation.

Two new data types have been now provided: (**con** and **ptr**) to increase the "agility" of the system.

The reserved word **:execute** was implemented to allow the programmer to take advantage of the different system modes while within a **routine**. The previous version only allowed the execution of different modes to take place from the terminal.

Potential System Improvements

The first needed improvement of the current system is the development of a file system to allow the programmer the ability to create generic data structures and maintain many copies of those structures within the system. As an example, consider an application which monitors the flight paths of all planes within some radius of a radar station. A generic data structure to maintain information about a flight path for a particular plane (e.g. position, altitude, heading, speed, etc.) could be easily created using a **VAR-O**, but the only way to have a set of data corresponding to each plane in the airspace is to create many **VAR-O**'s with the same overall format.

The National Bureau of Standards, has implemented a file system which incorporates some of these ideas, however, that file system was not implemented at Lehigh because it is has been found to be very inconvenient to use, and does not allow the programmer the ability to access data from more than one of the structures at any given time. We feel that further research is needed to better define the functionality of a file system.

Another possible improvement to the system could be the creation of low-level drivers to interface the SBE-200 computer system with a personal computer or more generally, a modem. This would allow the programmer to take advantage of the large amount of software available on most personal computers to generate (or collect) data. That data could then be made available to the control system for processing through this interface.

A final suggestion for improving the Lehigh version of RCS is perhaps the most important. The human-interface must be expanded to allow for a variety of peripherals in addition to the keyboard (such as a "mouse," a voice

recognition system, a voice synthesizer, etc.). The development of graphics capabilities will play an essential role in the creation of that interface, since the way in which the system is presented to the user directly influences the programmability of the system itself.

Final Comments

The applications to which the real-time control system described by this thesis can be applied are many. The National Bureau of Standards is currently demonstrating the system's extensive capabilities by controlling a robot in a factory floor environment. The future intent at Lehigh University is to again take advantage of the close relationship maintained with NBS and also develop a robotic application.

The author would like to acknowledge the National Bureau of Standards and Lehigh University for their committment to the advancement of technology by supporting this research project.

BIBLIOGRAPHY

Albus, J.S., A.J. Barbera, R.N. Nagel, "Theory and Practice of Hierarchical Control," Twenty-Third IEEE Computer Society International Conference, pp. 18-38, September, 1981.

Barbera, Anthony J., M.L. Fitzgerald, James S. Albus, Leonard S. Haynes, "RCS: The NBS Real-Time Control System," Proceedings of Robots 8 Conference, Detroit, Michigan, June, 1984.

Barbera, Anthony J., M.L. Fitzgerald, "Recommended Procedures for the Design and Implementation of Real-Time Control Software," Robotics Support Project for the Air Force ICAM Program, a Final Report, 1982.

Barbera, Anthony J., M.L. Fitzgerald, James S. Albus, Leonard S. Haynes, "A Language Independent Superstructure for Implementing Real-Time Control Systems," Proceedings of International Workshop on High-Level Computer Architecture 84, Los Angeles, California, May 1984.

Kirschbrown, Richard H., Richard C. Dorf, "KARMA - A Knowledge-Based Robot Manipulation System: Determining Problem Characterstics," Proceedings of Robots 8 Conference, Detroit, Michigan, June, 1984.

Nagel, R.N. (Ed.), "Manufacturing engineering for tomorrow's needs, Proceeds of SME Engineering Foundation Conference, August, 1981.

VITA

Mark Del Giorno was born in Baltimore, Maryland on January 26th, 1963 to Donald L. and Martha B. Del Giorno. He graduated cum laude in June, 1984 from the University of Delaware where he received his Bachelor of Science in Electrical Engineering. While attending the University of Delaware, Mark received several awards, including the Electrical Engineering Faculty Award; the Engineering Alumni Association Award; and the William H. Paynter Scholarship for Academic Achievement. He was also elected a member of several honorary societies: Tau Beta Pi, a national engineering honor society, Pi Mu Epsilon for mathematics and Eta Kappa Nu for Electrical Engineering.

He is now enrolled in the Manufacturing Systems Engineering Program at Lehigh University and expects to receive his Master of Science in January, 1986.